7. Sorting

- Motivation
- Algorithm Analysis
- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Partition Algorithm
- Quick Sort Analysis
- Bucket Sort
- Radix Sort
- Sorting Summary

Motivation

- Sorting data is one of the oldest and best studied problems in computer science.

- How can we put N data fields in ascending or descending order based on a given data field.

- Key Issues are:
  - How much time will the algorithm take?
  - How much data storage will be needed?
  - Will this algorithm work for all data types or data orderings?

- A wide variety of algorithms have been invented for sorting with lots of pros/cons.

Algorithm Analysis

- To compare two algorithms it is helpful to know how many instructions are executed to process N data elements.

- For example, to total N integers we could use:

```
int sum=0;
for(i=0;i<N;i++)
    sum+=data[i];
```

- Here the inner most loop is executed N times.

- Similarly if we wanted to print the product of all possible pairs of numbers we could use:

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        cout << data[i] * data[j] << endl;
```

- Here the inner loop will execute N^2 times.

Algorithm Analysis

- Often the loops are more complex.

```
int count=0;
for(i=0;i<N;i++)
    for(j=i;j<N;j++);
        count++;
```

- Here the inner loop executes N + N-1 + ... + 2 + 1 times, which equals (N-1) * N/2 $= N^2/2 + N/2$

- This is less than the N^2 in our previous example, but the difference is not significant.

- Ignoring the constants and lower order terms, we say both algorithms are "Order N^2" denoted as $O(N^2)$.

Algorithm Analysis

- Algorithms that use divide and conquer (like binary search) can execute a loop in less than N steps.

```
int num=N;
while(num>0)
   num=num/2;
```

- Here the inner loop will execute $O(\log_2 N)$ times.

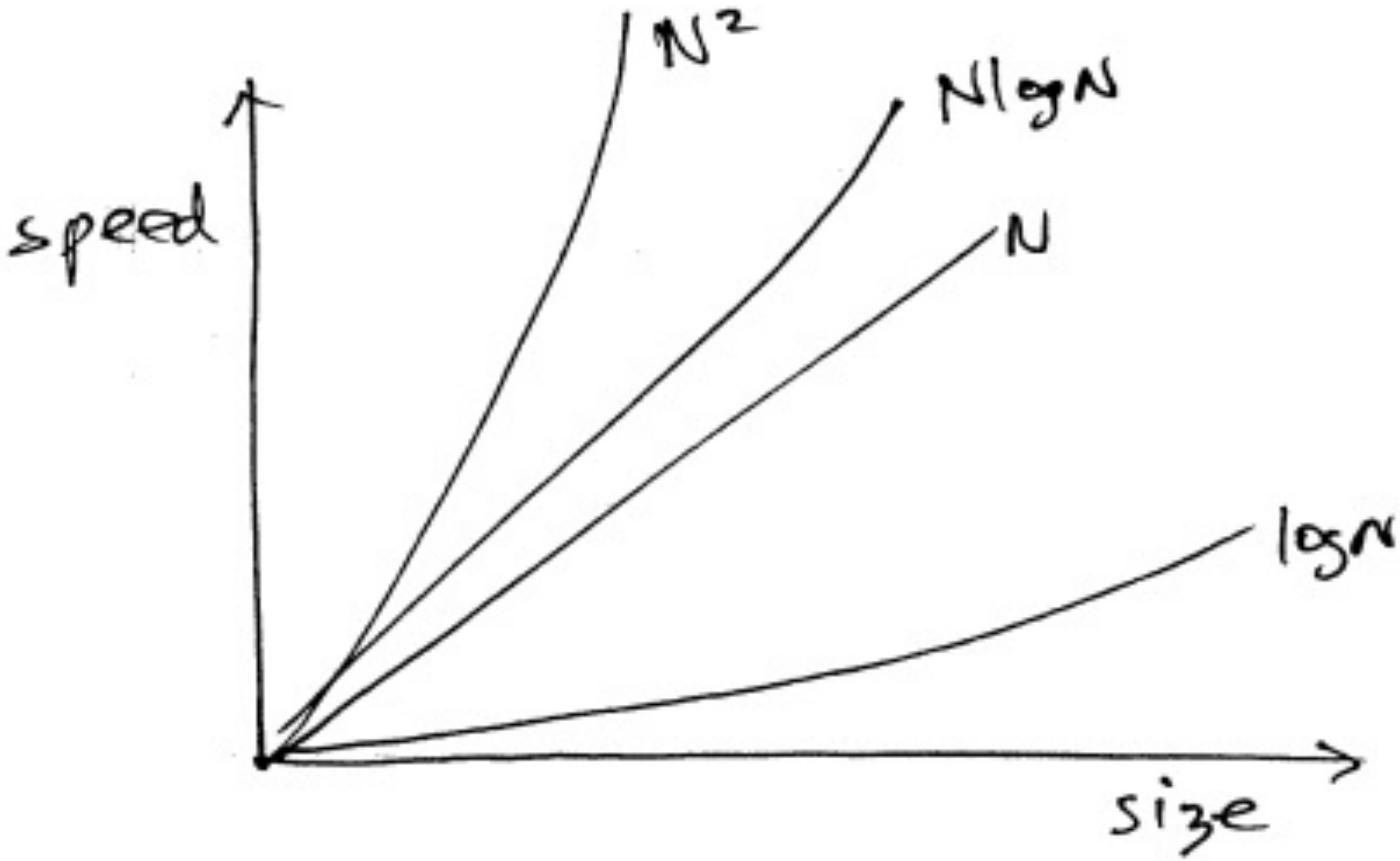
- Another example combines a $O(N)$ loop with a $O(\log_2 n)$ loop.

```
for(i=0;i<N;i++){
   int num=N;
   while(num>0)
      num=num/2;
}
```

- Here the inner loop will execute $O(N \log_2 N)$ times.

Algorithm Analysis

- $O(\log_2 N) < O(N) < O(N\log_2 N) < O(N^2)$

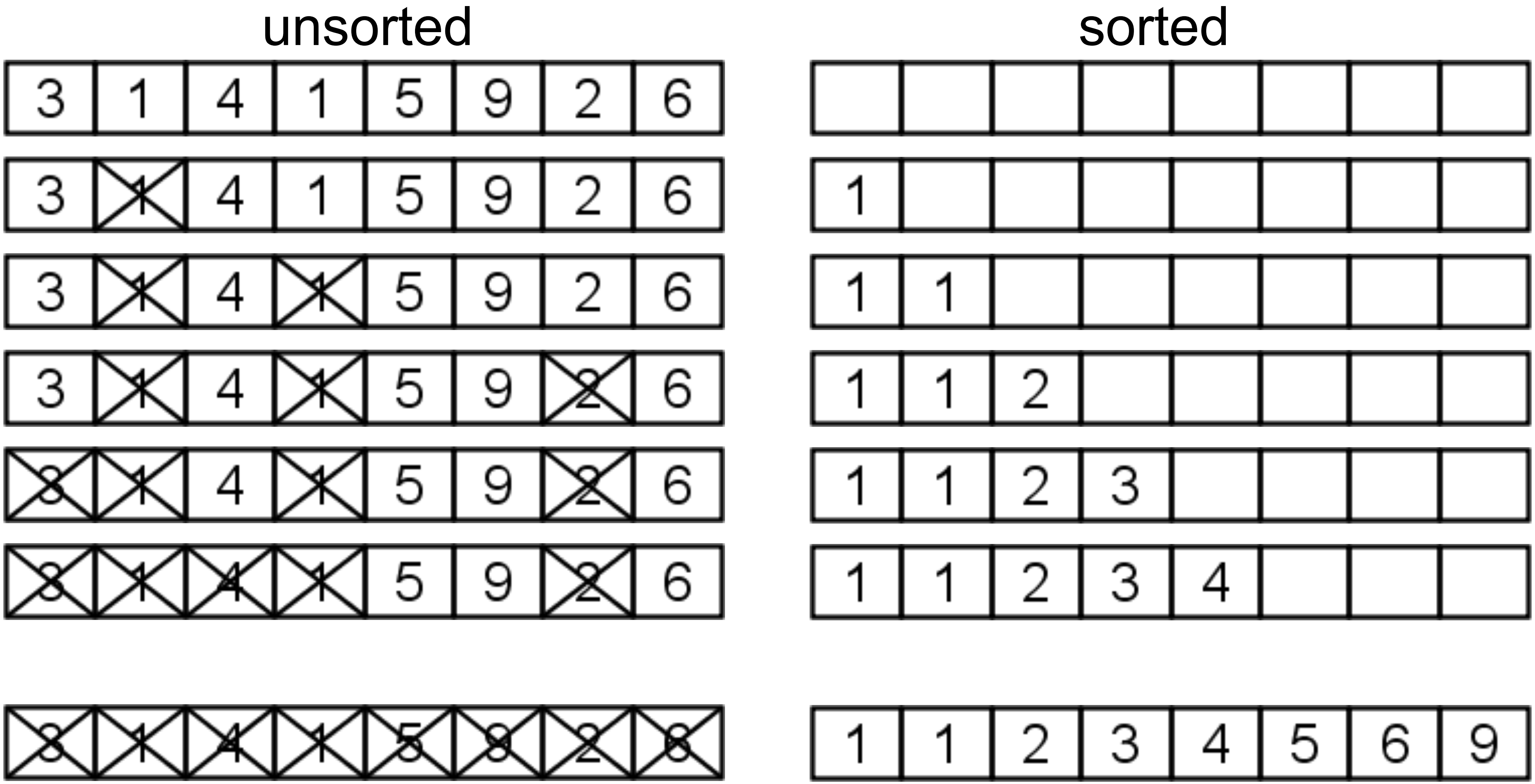


- Eventually you will see algorithms that are worse than $O(N^2)$ but these are never used for sorting data.

Selection Sort

- Selection sort is a simple but slow sorting algorithm.
- The idea is to iteratively select the smallest value from an unsorted array, and put this at the end of a sorted array.
- Loop N times
  - Select smallest value in unsorted array.
  - Mark data as "taken".
  - Store data at end of sorted array.
- When we are done, the unsorted array will be empty, and the sorted array will have N values in ascending order.

Selection Sort



- The sorted array will be filled after N iterations of selections.
- Since only N locations are used, we can implement this using just one array N long.

See Selection Sort Code

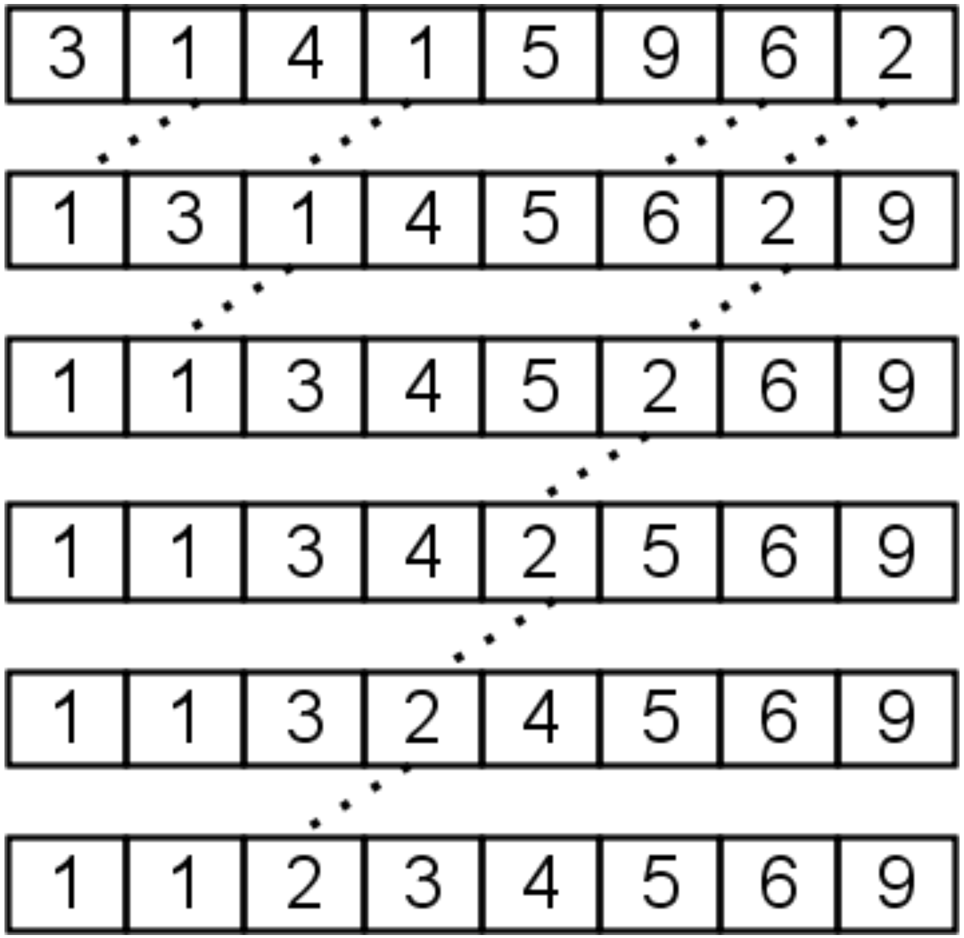
Selection Sort

- Looking at the selection sort code we see two nested loops that do the work.
- Hence the Algorithm is $O(N^2)$.
- Does it get faster/slower if the data is reordered?
- In this case it still takes $O(N^2)$ steps to sort the data even if the data is sorted in the first place.
- Hence the best case = worst case = average case = $O(N^2)$

Bubble Sort

- Bubble sort is another well known but slow algorithm.
- The idea is to iteratively compare adjacent data values in array and swap them if they are out of order.
- On each pass over that data, the largest value "bubbles" to the right, and smaller values shift one position to the left.
- Loop N times over array
  - Loop over N data values
  - Compare adjacent values
  - Swap if needed
- Stop when no swaps take place.

Bubble Sort



- Notice that small values move slowly to the left, and large values move quickly to the right. (more than 1 space).

- In this case, it only took 2 phases to sort the data.

See Bubble Sort Code

Bubble Sort

- Looking at the bubble sort code we again see two nested loops so this code is $O(N^2)$ in general.
- As we saw, data that is almost completes fewer iterations.
- In the best case, when the data is already sorted, we stop after only 1 pass.
- Hence, in the best case bubble sort is <u>O(N)</u>
- In the worst case when data is in reverse order we need $O(N^2)$ steps to sort.

Insertion Sort

- Insertion sort is another classic sorting algorithm.
- This is the approach we used to keep a linked list in sorted order.
- The idea is to search a sorted section of data to find the position new data should be inserted.
- If we start with 0 sorted elements and insert one element N times, the whole array will be sorted.
- Loop over N unsorted values
  - Pick next value to insert
  - Loop over ~N sorted values
    - Find insertion location
    - Store data in location

Insertion Sort

unsorted	sorted
3 1 4 1 5 9 2 6	
~~3~~ 1 4 1 5 9 2 6	3
~~3~~ ~~1~~ 4 1 5 9 2 6	1 3
~~3~~ ~~1~~ ~~4~~ 1 5 9 2 6	1 3 4
~~3~~ ~~1~~ ~~4~~ ~~1~~ 5 9 2 6	1 1 3 4
~~3~~ ~~1~~ ~~4~~ ~~1~~ ~~5~~ 9 2 6	1 1 3 4 5
~~3~~ ~~1~~ ~~4~~ ~~1~~ ~~5~~ ~~9~~ 2 6	1 1 3 4 5 9
~~3~~ ~~1~~ ~~4~~ ~~1~~ ~~5~~ ~~9~~ ~~2~~ 6	1 1 2 3 4 5 9
~~3~~ ~~1~~ ~~4~~ ~~1~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~	1 1 2 3 4 5 6 9

- Notice that we ended up moving the data in the sorted array to the right to make room for new values.
- What happens if input is already sorted?

See Insertion Sort Code

Insertion Sort

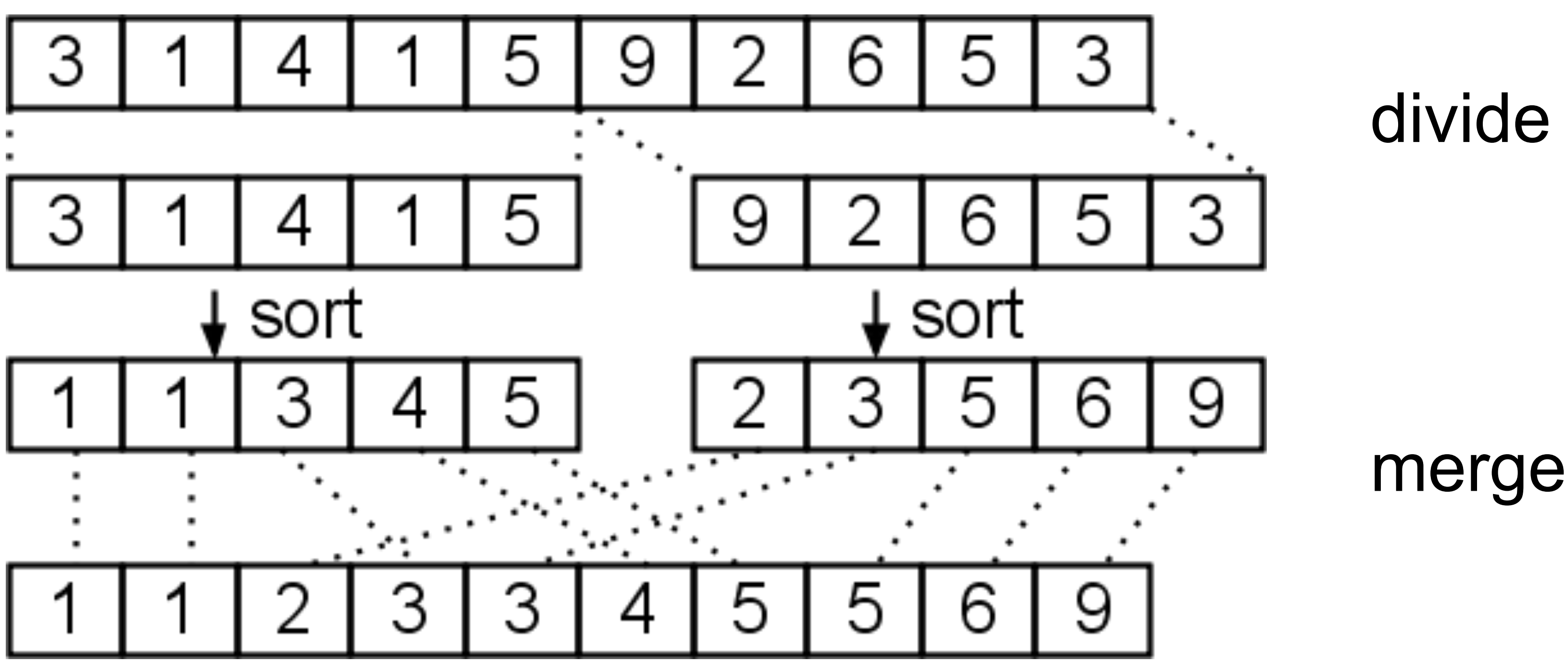
- Looking at the insertion sort code we see two nested loops so this code is $O(N^2)$ on average.

- The best case occurs when the data is already sorted, then it only takes $O(N)$ steps.

- Notice that we avoided two data arrays by storing the sorted data on the left and the unsorted data on the right of the input array.

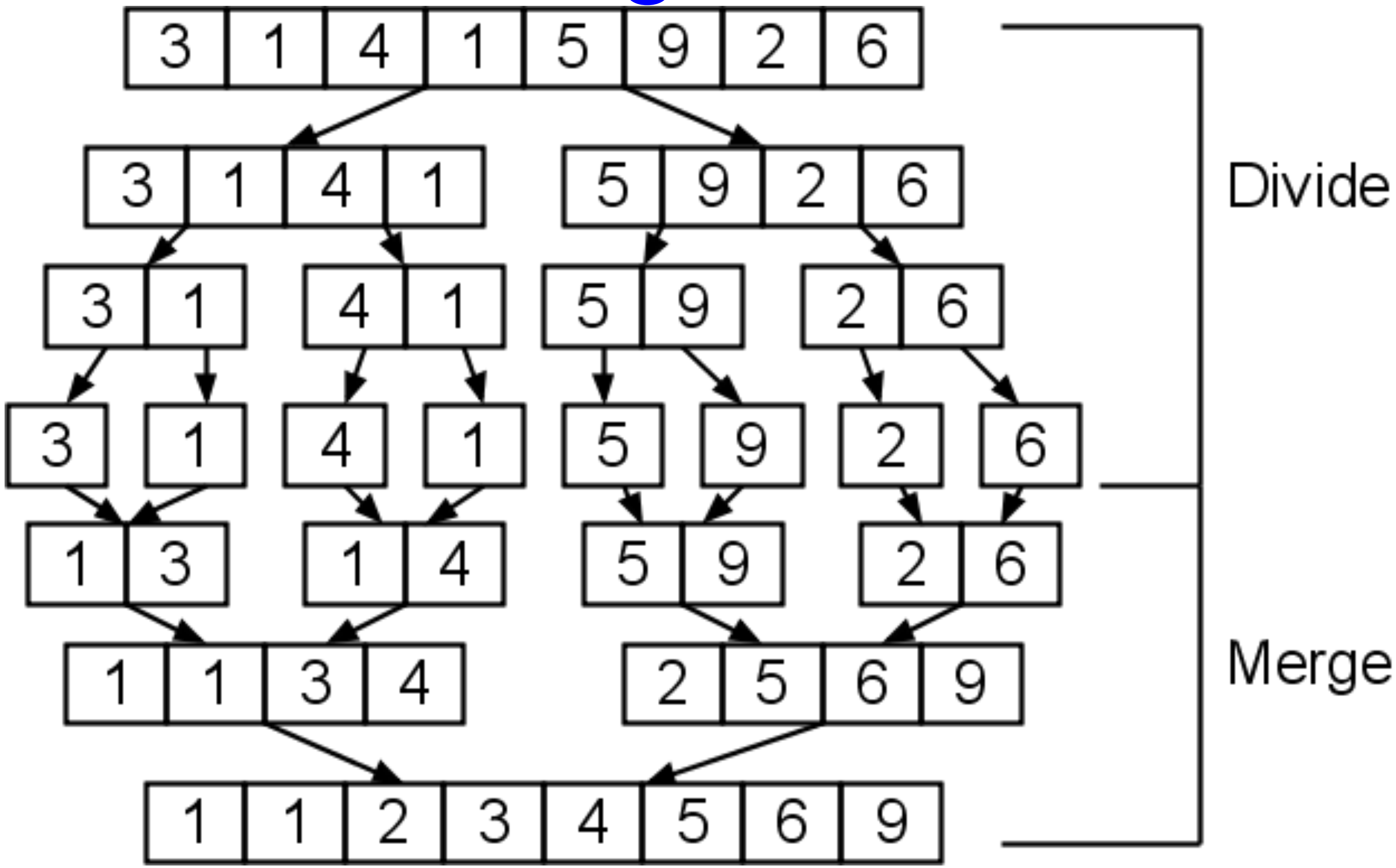
Merge Sort

- Merge sort is a classic divide and conquer algorithm.
- The idea is to:
  - Divide input into two parts.
  - Sort each half recursively.
  - Merge values into sorted array.



- We can use recursion to sort the N/2 values prior to merging.

Merge Sort



No real work is needed to divide the input array into two halves, we just calculate midpoint of array and call mergesort recursively. The merge requires a L->R scan of both arrays and the data must be copied into an output array.

See Merge Sort Code

Merge Sort

- If we define the amount of work to sort data as S(N) then:

$S(1) = 1$ <- Single element sort

$S(N) = N + 2 * S(N/2)$

merge sort both halves

$S(N) = N + 2 * (N/2 + 2 * S(N/4)$

$= 2N + 4 * S(N/4)$

$= 3N + 8 * S(N/8)$

$= kN + 2^k * S(N/2k)$

When $N=2^k$, $k=\log_2 N$, substituting we get

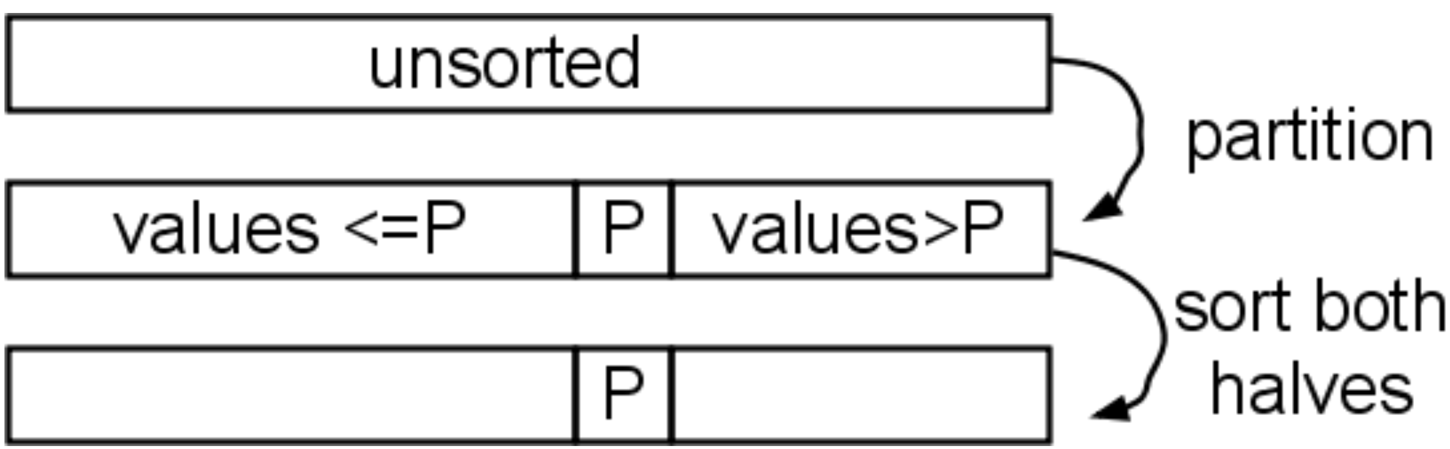
$S(N) = N * \log_2 N + N * S(1)$

$S(N) = O(N\log_2 N)$

$\therefore$ - Much faster than $O(N^2)$

Quick Sort

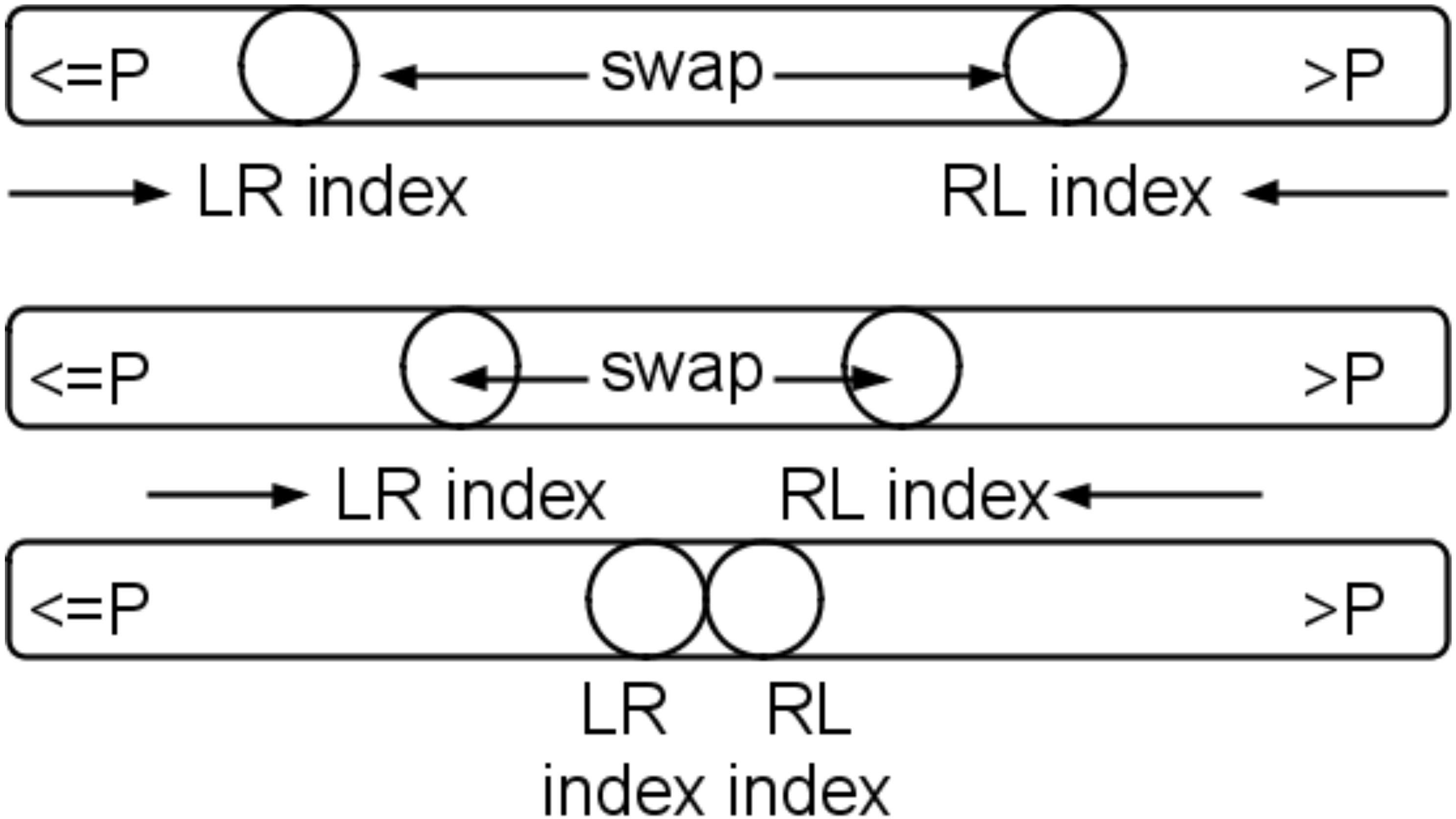
- Quick sort is another divide and conquer algorithm that is well known for being fast. (hence the name)
- The idea is:
  - Partition array with small values on left and large values on the right.
  - Sort each half recursively.
  - Result is sorted array.
- There is no work merging the data, but we need to look at each value to partition.



- Result is a sorted array. (Since we know that no value on L partition belongs to R side.)

Partition Algorithm

- Fast algorithm by Sedgewick scans LR for value > partition, then scans RL for value < partition, and swaps their values.



- The scan/swap algorithm stops when the RL index is less than the LR index.
- It looks at each value, so it is O(N)

Partition Algorithm

- Tricky question is how to pick a pivot value p to evenly divide the input array into two parts.
- Common choices are to choose either the leftmost, rightmost, or middle value in input array.
- Once chosen, we just "hope for the best" and partition based on this value.

3	1	4	1	5	9	2	6	input array
0	1	2	3	4	5	6	7	
1	1	2	**3**	4	5	9	6	leftmost=3
	3		p		4			
3	1	4	1	5	2	**6**	9	rightmost=6
		6				p	1	
1	**1**	3	4	5	2	6	9	middle=1
1	p			6				

- Another choice would be to use the <u>median</u> of leftmost, rightmost, middle (3 in this case)

See Quick Sort Code

Quicksort Analysis

- Consider the best case where pivot divides the problem in half at each step.

$$S(1) = 1 \quad \text{<- Single element sort}$$

$$S(N) = \underbrace{N}_{\text{partition}} + 2 * \underbrace{S(N/2)}_{\text{sort both halves}}$$

$\therefore S(N) = O(N\log_2 N)$ - Like Merge sort

- In practice, the partition is close enough to optimal that quick sort has run times that are almost $O(N\log_2 N)$.
- In the worst case we can end up with 1 value in one have and N-1 values in the other half (if we chose largest or smallest as partition)

Quicksort Analysis

$$S(0) = S(1) = 1$$

$$S(N) = S(N-1) + S(1) + N$$

$$= S(N-2) + 2 * S(1) + N + N - 1$$

$$= S(N-k) + k * S(1) + \sum_{i=0}^{k-1} (N-i)$$

when k=N, the first term becomes S(0)=1

the second term becomes N*S(1)=N

$$S(N) = 1 + N + \sum_{i=0}^{k-1} (N-i)$$

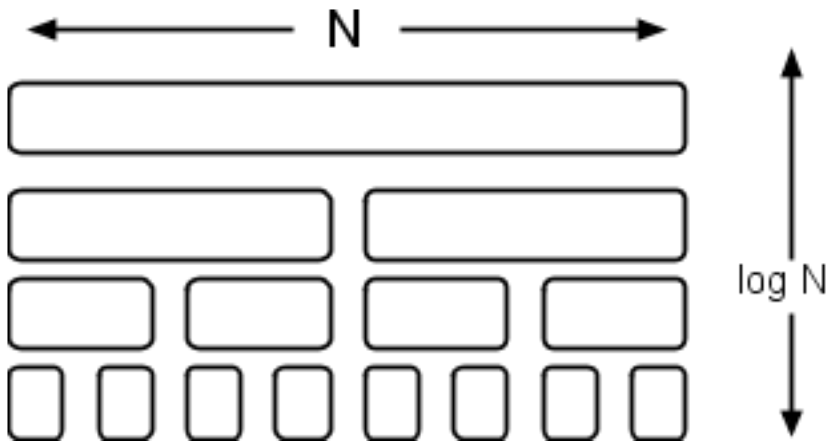
$$= 1 + N + (N+1) * N/2$$

$$= 1 + N + N^2/2 + N/2$$

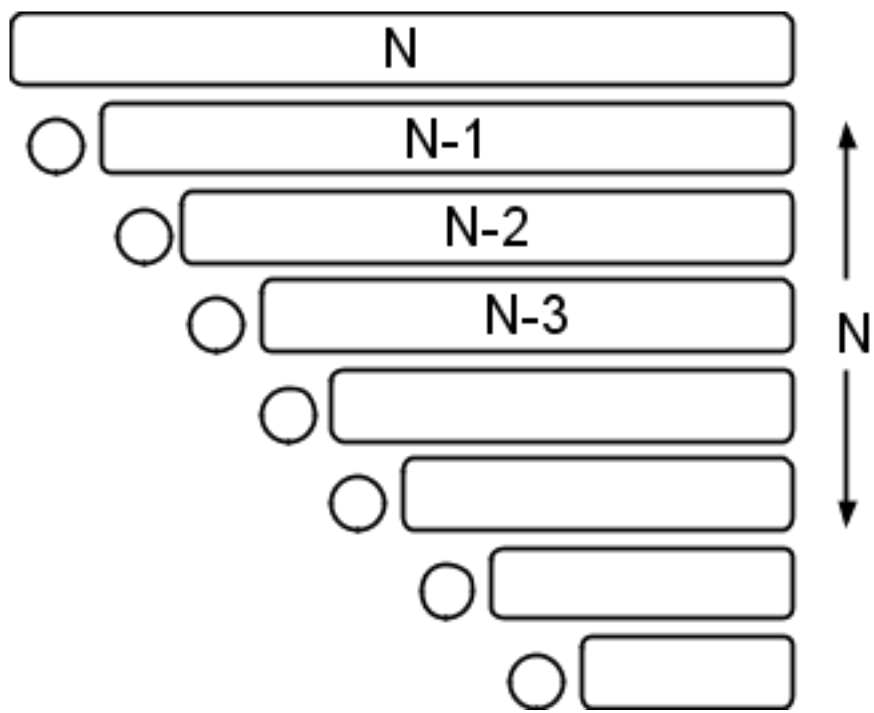
$$\therefore S(N) = O(N^2)$$

Quicksort Analysis

<u>Best case</u>:
$O(N\log_2 N)$



<u>Worst case</u>:
$O(N^2)$



Bucket Sort

- When we know the input data has a limited range of values we can use bucket sort.
- Idea is to do one pass over the data and <u>count</u> the number of times each value occurs in the input (hence this is often called counting sort).
- Output is generated by looking at the count array and printing the appropriate number of each value.
- If we are lucky this can be <u>very</u> fast.

Bucket Sort

- Consider sorting the first 30 digits of PI.

input: 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3
2 3 8 4 6 2 6 4 3 3 8 3 2 7 9

counts:

0	1	2	3	4	5	6	7	8	9
0	2	4	7	3	3	3	2	3	4

output: 1 1 2 2 2 2 3 3 3 3 3 3 3 4 4 4
5 5 5 6 6 6 7 7 8 8 9 9 9 9

See Bucket Sort Code

Bucket Sort

- Algorithm needs only one pass over data to count values, and one pass over counts to output results.
- The counting phase is O(N) because there are N values to examine.
- The output phase is O(M) where M is the size of the counting array.
- $\therefore$ Bucket Sort is O(N+M)

- When the range of values is small (eg. digits of PI) this is almost O(N), which is amazingly fast.
- When the range of values is large (eg. ID's) the M term will be much larger than N and bucket sort will be very slow.

- Bucket sort doesn't work well for floats for strings, so this is special case algorithm.

Radix Sort

- Radix sort is another special purpose algorithm that can be very fast for certain data.

- The idea is to look at digits in number (or letters in string) one at a time, and sort by that value.

- Repeating this from least significant digit to most significant digit (or letter) we end up with sorted data.

- Key is making sure the order of data values that "tie" remain unchanged as we sort each digit.

- For numbers we could use I0 linked lists to store intermediate data, for strings we would need 26 lists (more if upper/lower case are considered).

Radix Sort

3 1 4	3 2 3	3 1 4	1 5 8
1 5 9	3 1 4	3 2 3	2 6 4
2 6 5	2 6 4	3 2 7	2 6 5
3 5 8	2 6 5	3 3 8	3 1 4
9 7 9	8 4 6	8 4 6	3 2 3
3 2 3	3 2 7	3 6 8	3 2 7
8 4 6	3 5 8	1 5 9	3 3 8
2 6 4	3 3 8	2 6 4	3 5 8
3 3 8	1 5 9	2 6 5	8 4 6
3 2 7	9 7 9	9 7 9	9 7 9
input data	sort	sort	sort

- In this case with 3 digit numbers we sorted by 1's column, then by 10's column and finally by 100's column.
- A similar approach can be applied to strings working right to left sorting by ascii codes.

Radix Sort

- Algorithm needs to make one pass over data for each digit/letter in input.
- If we let R be number of digits/letters then radix sort is O(R*N)
- Clearly it is better to sort 100 3-digit numbers using radix sort than to sort 3 100-digit numbers.
- Depending on the data structures used to store values as we sort, radix sort may take more space than traditional "in place" sorting algorithms.

Sorting Summary

	Best	Ave.	Worst
Selection Sort	N^2	N^2	N^2
Bubble Sort	N	N^2	N^2
Insertion Sort	N	N^2	N^2
Merge Sort	$N \log N$	$N \log N$	$N \log N$
Quick Sort	$N \log N$	$N \log N$	N^2
Bucket Sort*	$N+M$	$N+M$	$N+M$
Radix Sort*	$R \cdot N$	$R \cdot N$	$R \cdot N$

* These algorithms have restrictions on input data type/values.